Natural language and programming: designing effective environments for novices

Judith Good, Kate Howland
Department of Informatics
University of Sussex
Falmer, UK
{J.Good, K.L.Howland}@sussex.ac.uk

***Abstract*— Given the current drive to teach computational concepts to all from an early age, we consider whether traditional programming languages are truly necessary, or whether natural language might be a suitable medium for program generation and comprehension, given its familiarity and ubiquity. We conducted an empirical study on the use of natural language for computation, and found that, although it provides support for understanding computational concepts, it introduces additional difficulties when used for coding. Following a design study with target users, we distilled our findings into a series of design guidelines for novice programming environments that incorporate natural language. These guidelines drove the design of Flip, a bimodal programming language for young people's game creation activities. An empirical study examined the extent to which these embodied design guidelines support ease of use and an understanding of computation. The guidelines have potential both for analysing the usability of existing novice programming environments, and for designing new ones.**

Keywords—novice programming languages; natural language; design; empirical evaluation

I. INTRODUCTION

The drive to make programming accessible to a broad range of individuals is gaining momentum in schools, with countries such as England making computer science a mandatory part of the curriculum [2]. Things are similar in professional contexts, with computation being increasingly central to almost every discipline. Since domain experts are best placed to understand the needs of their sector, it makes sense that they should be able to program the devices they use to accomplish their tasks. Recent developments in society as a whole, with entertainment and leisure activities incorporating digital devices, also suggest that end users should be able to customize devices as they see fit. All of these scenarios require at least a basic ability to write simple programs, which has led to an increasing focus on programming languages that are both easy to learn and to use.

At the same time, the current drive around computation more broadly calls into question the role of programming languages. In a school context, helping children understand the fundamental concepts that underpin computation may not necessarily require complex programming languages. Similarly, at a societal level, allowing end users to adopt computational approaches to tasks and problems may not need a general purpose programming language.

In all cases, it's about ensuring that the unnatural or complex program syntax of traditional programming languages is not a barrier to the understanding of computation, or the use of computational techniques. In this context, one obvious question to ask is why people can't simply program using natural language? It is well established that programming language syntax is a major stumbling block for novices [6]. Given that both children and adults already use natural language to express ideas and concepts, it could provide a simple solution to syntax problems, while eliminating the need to learn a new language. This is by no means a new question: similar arguments have been made as far back as 1966 [13]. Although natural language processing may have lacked sufficient power in the past, languages such as Inform 7 make natural language programming a more realistic possibility [9].

In this paper, we firstly examine the viability of natural language based programming languages through an empirical study[1]. We then describe a design study that allowed us to synthesise our findings into a set of guidelines for the use of natural language in programming. These guidelines, which will be relevant to the design of languages for non- or novice programmers, were implemented in Flip, a bi-modal programming language for young people. Flip was evaluated in a series of studies, and we report on two evaluations which consider the extent to which the design guidelines, as embodied in Flip, were able to provide effective support for novices.

II. BACKGROUND

A. Game creation and programming

Within the broader "programming for all" context, our motivation for designing a "natural" programming language stems from our longstanding research into game creation for young people [3, 4, 11, 12], carried out primarily using the Neverwinter Nights 2 (NWN2) Electron toolset (shown in Fig. 1). The toolset allows young people without specialist skills to quickly and easily begin building a game, and is highly motivating, as the games created are similar in appearance to professionally developed 3D commercial games.

Game creation allows young people to become producers of technology, introduces them to computational concepts and

[1] Ethical approval was obtained from the University of Sussex's ethics committee for all of the studies described in this paper.

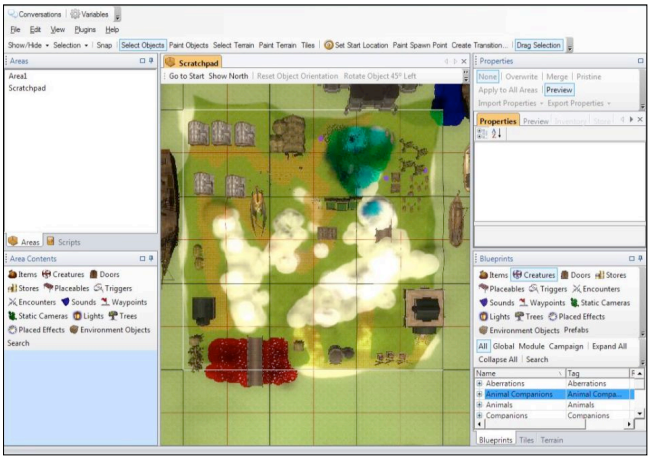

Fig. 1 Neverwinter Nights 2 Electron Toolset Interface

helps them develop programming skills. As young people's games become more complex, they find themselves needing to script events in order to make their game work as they wish. For example, they may wish to reward the player with treasure when they slay the dragon, or cause a wizard to vanish when they cast a spell. Unfortunately, the Electron toolset uses NWScript, which is based on C and has a similarly complex syntax. In our previous game making workshops, involving over 350 young people in total, no participant was able to learn NWScript sufficiently well to script their own events and, instead, had to rely on the workshop facilitators to translate their story ideas into scripts.

With the complex syntax of NWScript acting as a barrier to the underlying computational concepts, we began to explore ways of allowing young people to engage with computation more directly. We had observed that young people used natural language quite accurately to describe what they wanted to happen in their games, although event descriptions were sometimes underspecified, requiring additional prompting to be complete [5]. The finding that these errors were primarily errors of *omission* (failing to include relevant parts of the description) rather than errors of *commission* (including erroneous elements in the description) is in line with much earlier work on natural language descriptions of code [7, 8].

We aimed to use young people's existing narrative understanding of game events and their ability to describe events in natural language in order to design a new language which would allow them to engage with basic computational concepts. As such, natural language programming was a promising area to explore.

B. *Initial explorations of natural language for programming*

An initial exploration, reported in full in [1], investigated whether young people could use natural language to write rules that correctly described behavior occurring in a computer game. Sixty four pupils aged 11-12 (35 female, 29 male) took part in the study. They were asked to play a game, created specifically for the study, which contained a number of scripted encounters, embodying increasingly complex computational structures. For each game encounter, pupils were asked to "write a rule" that would produce the behaviour they had just experienced. Errors in their rules were categorized using a coding scheme based on the error analysis used in [2].

Overall, rule correctness was low: only 21% were fully correct, with a further 35% being partially correct. Errors of omission accounted for 74% of all errors, i.e., rules were much more likely to be *incomplete* than *inaccurate*. The low proportion of inaccurate rules is likely due to the impossibility of syntax errors: because the rules were "interpreted" by humans, multiple syntactic variations could be considered correct, provided they were semantically correct. On the other hand, because the exercise was paper-based, pupils were unable to test their rules, meaning they received no feedback on whether their rules functioned as they had intended.

From a programming language design perspective, these findings suggest that support designed to minimize syntax issues is not sufficient. The environment should also provide robust support for errors of omission, ideally before compiling and testing. This could take the form of highlighting, where possible, rules which contain missing elements, or providing a "read back" function to allow users to check that their code matches with their intentions.

In the following section, we describe an exploratory study designed to further our understanding of the use of natural language for novice programming. We go on to describe a design study in which we tested developing hypotheses about how natural language could be used. We then draw together the implications from the two studies, and the study just mentioned, into a set of design guidelines around the use of natural language for novice programming environments.

III. Inform 7 Study

In order to gain an in-depth understanding of how natural language programming languages are used by novices, we conducted an empirical study using Inform 7 (inform7.com), a fully functional natural language programming environment for the creation of interactive fiction (digital text-based adventures). Inform 7's aim is to be accessible to non-programmers, with code designed to read like English.

To examine issues of ease of use, learnability and comprehension, we ran a three hour workshop in which we observed non- and novice programmers creating pieces of interactive fiction using Inform 7. We were interested in investigating whether errors and misconceptions occur and, if so, understanding the nature of these errors and misconceptions, particularly those relating directly to the use of natural language for programming.

A. *Method*

Nine university students (8 female and 1 male, aged 18-42) took part in the workshop, advertised via the university's English department so as to attract individuals with skills in creative writing, but limited programming experience.

Following an introduction, participants were shown how to play a completed Inform 7 game to allow them to experience interactive fiction from the "reader/player's" perspective and introduce them to common player commands. We then

demonstrated how to create an Inform 7 story from scratch to introduce the commands needed to create interactive fiction.

Participants were given two "cheat sheets" showing common commands and syntax for *playing* and *writing* a game respectively. They were also provided with a broad outline of a story to create, and asked to work in groups of two or three to create the story. The groups were then asked to create their own piece of interactive fiction. We finished with a debriefing and informal group feedback session.

Audio recordings were transcribed and paired with the screen recordings, and coded to identify code errors and participant misconceptions with respect to the Inform 7 language. This allowed us to develop an initial, empirically derived, taxonomy of errors and misunderstandings.

B. Results

To support reader understanding of the errors and misconceptions described, Fig. 2 shows an annotated sample of Inform 7 code. The first line is a standalone phrase, and the following two lines are a rule consisting of a rule preamble and a phrase that executes when the rule preamble is met. Phrases and rules must follow a specific syntax, or 'pattern'. In Fig. 2, the first phrase has the following pattern, in which the articles are optional: (The) [object] is (a) [description].

The error/misconception taxonomy is shown below:

1. Confusion between natural language as a programing language and 'free' natural language (strings)
- *1a) Placing rule and phrase keywords within strings*
- *1b) Placing descriptive text outside of strings*
- *1c) Wrongly assuming string rules that do not exist*

2. Errors using natural language as a programming language
- *2a) Using synonyms in place of rule and phrase keywords*
- *2b) Incorrect syntax of rules and phrases*
  - *2c.i) Incorrect ordering*
  - *2c.ii) Adding additional words*
  - *2c.iii) Omitting rule and phrase keywords*
- *2e) Using one keyword in place of another*
- *2f) Problems with object names*
  - *2f.i) Typographical errors*
  - *2f.ii) Inconsistent typing of object names*
- *2g) Wrongly assuming syntax rules that do not exist*

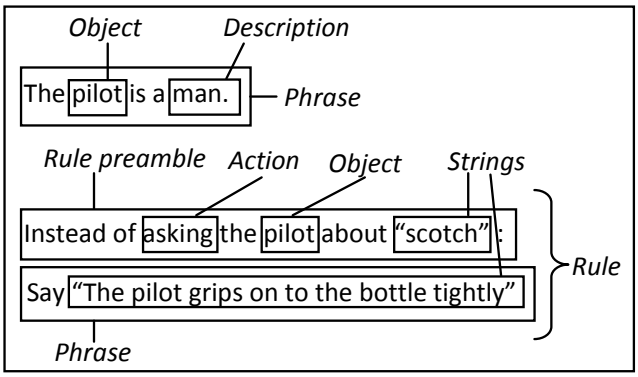

Fig. 2 Annotated Inform 7 Phrase and Rule

All participants found Inform 7 very challenging, and struggled to implement their story ideas and develop a working piece of interactive fiction. Program code must be written using very particular syntactic structures and keywords, and although this syntax is documented in help files and online manuals, there is no dynamic support for syntax when writing code. As a result, syntax errors frequently stopped games from compiling, preventing users from being able to test their games.

A primary source of confusion was between the use of natural language as a programming language (i.e. Inform 7 commands) and 'free' natural language (i.e. strings). Participants sometimes placed rule keywords and phrases within strings rather than outside them (1a), for example, including the keyword 'say', used to print to screen, within the string to be printed, i.e. <Instead of asking the pilot about scotch: "The pilot says 'hands off' and grips on to the bottle tightly">.

The other main category of error concerned the use of natural language as a programming language. Semantically similar words were frequently used in place of the correct keyword (2a). For example, a participant who wanted players in the 'Crash Site' area to be able to enter the plane fuselage by typing "go inside", wrote <the Crashed Plane Interior is *in* the Crash Site>, whereas the correct code is <the Crashed Plane Interior is *inside from* the Crash Site>, admittedly, a not very English-like syntax. Participants also inserted extraneous words into rules and phrases (2c.ii). A participant who wanted a player who was currently 'somewhere in the desert' to enter an oasis by typing 'go west' wrote "the oasis is to the west of somewhere in the desert" whereas the correct syntax is "the oasis is west of somewhere in the desert".

When a compilation error occurs, Inform 7 tries to offer helpful feedback. Although designed to simulate a conversation, many participants found the verbose style of these error messages confusing rather than helpful. The error message in Fig. 3, generated in response to the error identified in the above paragraph (i.e. writing "to the west of" rather than "west of"), was of no help in allowing the authors to identify their small syntax error. In cases where the error message helped participants identify the problem, they were often unsure how to fix it. In a few cases, the pseudo-conversational style of the error messages led to feelings of frustration at not being able to reply, with one participant asking her group members, "So how do you say 'Yes, actually, you *have* misunderstood me!'?".

C. Discussion

Overall, the use of a natural language programming language did not seem to benefit novices: paradoxically, the

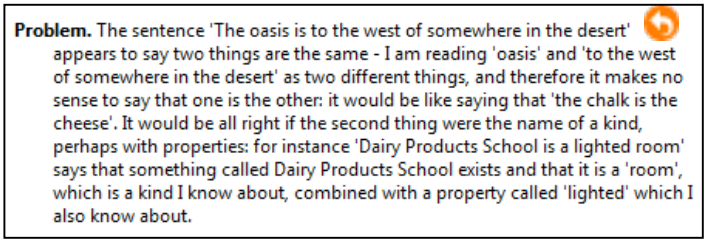

Fig 3 Verbose error message

errors and misunderstandings observed seemed to have stemmed from the very features designed to provide support, i.e. program code that 'reads like English'. Users struggled with the distinction between "natural language as natural language" and "natural language as programming language", and were unsure when the language must be constrained, and when syntactic variations are allowable.

Many of these issues reflect essential differences in the way that humans and computers typically "use" language. Languages allow for dialogue between two entities, typically humans (although in this case, between a human and a computer). Natural languages have inbuilt redundancy, affording multiple ways of expressing a single idea, and humans are adapt at using natural language to generate synonyms at word and phrase levels. Relatedly, they can correctly interpret multiple syntactic variations of a semantic idea. Programming languages, however, lack this redundancy, and compilers are not designed to deal with it: a synonym of a keyword means absolutely nothing.

Using natural language as a programming language (i.e. for communication from human to computer) highlights this mismatch between the human ability to generate syntactic variations of a semantic idea, and the computer's ability to understand only one of these. Furthermore, having to write Inform 7 code from scratch, rather than by selecting from a set of words/phrases that the interpreter can understand, multiplies the possibility of syntactic variations.

Communication in the opposite direction, from computer to human, raises other issues. Given the human ability to correctly interpret syntactic variations, the actual phrasing of a communication, such as an error message, is less important as long as it is comprehensible (which is, admittedly, hard to operationalize). However, messages from the compiler which are designed to be "human-like" set up a false expectation that it is capable of more extensive human-like communicative exchanges. Indeed, such a situation may exacerbate the formation of "superbugs", where learners believe "there is a hidden mind somewhere in the programming language that has intelligent, interpretive powers" [10, p.25].

The difficulties experienced by Inform 7 users in using natural language for programming could potentially be addressed by increasing the levels of constraint and scaffolding. For example, constraint could be achieved by allowing users to select from a set of keywords or alternatively, a more advanced system could potentially interpret natural language more generously, e.g. not throwing up a compiler error when a user types "…description for…" rather than "…description of…". Furthermore, other problems might be avoided by ensuring that messages from the computer are comprehensible, and do not set up false expectations. Taken together, this suggests that although it may not be possible to use natural language for programming, it may be of use for program comprehensions.

IV. LOW-FIDELITY CARD PROTOTYPE DESIGN STUDY

Our findings from a previous study (described briefly in Section II.B), and the study described in the previous section, indicated that when novice programmers use freeform natural language to specify computational rules, a surprisingly high number of errors result. However, we hypothesised that the problems arose not because of the natural language aspect per se, but because of the lack of language constraint and support during the program construction process. We therefore wanted to explore whether giving novices a restricted set of language primitives with which to assemble computational rules would result in greater accuracy. Providing this form of syntactical support would avoid the problems of multiple semantically correct, but syntactically uninterpretable, rules, as was seen in the study with Inform 7.

We opted to use natural language for computational keywords, but images for objects and characters, given that they are represented graphically in the NWN2 environment. The aim was to provide the most direct mapping for users, eliminating the need to translate between a visual representation of a character or object, and its name.

A. Method

The design study was conducted in two different contexts: a classroom setting and a holiday workshop setting. In total, 20 young people took part in the study: a sample of 8 pupils from a school aged 11-12 who were involved in a gamemaking project in their IT class (2 female, 6 male), and 12 young people (aged 11-15, all male) who had elected to attend a game making workshop during their school holidays.

The study used a set of laminated cards of four types: *action* cards, *thing* cards (objects and characters), *connecting* cards (control logic such as *if*, *until*), and *description* cards (object state). The *thing* cards graphically depicted common characters and objects in the game world, while the words used on the other three card types were drawn from the corpus of natural language rule descriptions generated in the previous study. We also included a number of blank cards should participants wish to create new cards.

We were interested in determining the ease of both language comprehension and code generation. Participants were therefore first asked to read out the meaning of rules that had been constructed using the cards (25 in total, increasing in complexity). This was followed by a code generation activity in which we read out a statement (15 in total, again increasing in complexity), such as "The wolf attacks the player, but only if the player is carrying the treasure" and asked participants to use the cards to construct a rule.

B. Results

Overall, the majority of participants were able to construct correct rule descriptions using the cards. For the composition task, 76% of the rules were fully correct at the holiday workshop, while 77% of rules were fully correct in the school study.

There did, however, seem to be some confusion between different computational categories, particularly between states and actions. A state 'is open' was often used in place of an action 'opens'. Similarly, 'when' and 'if' were often used in place of each other.

C. Discussion

In comparison to the study in which pupils wrote rules in unconstrained natural language (section II.B), the percentage of correct rules was considerably higher, indicating that being able to choose from a limited vocabulary reduced errors substantially. This is not surprising in and of itself, however, it is interesting to note that while graphical languages require users to choose from a set of pre-existing blocks by necessity, most text-based languages do not, and instead require users to type their statements from scratch, introducing syntax errors in the process. While some suggest that graphical languages are superior to textual languages for novices [15], it may be that improvements in terms of ease of use result from the constrained nature of graphical languages, rather than their graphicacy per se.

Finally, the confusion between different types of computational constructs suggests that support should be provided to help users understand the specific category of computational construct.

V. SUMMARY OF FINDINGS

In our study of the ways in which young people use natural language to convey computational concepts (study 1, Section II.B), we found that eliminating syntax and compiler issues did not completely eliminate errors, however, the errors took the form of errors of omission, rather than errors of commission.

The Inform 7 study (study 2, Section III), investigating the use of a natural language-like programming language for code generation, suggested that unconstrained natural language introduces a number of significant issues. Some of these stem from confusion as to how language is being used in any given instance (i.e. is this "real" natural language, or "programming" natural language?). Even when the user is aware of the distinction, the human ability to generate multiple semantically identical phrasings comes up against the compiler's ability to understand only one of these phrasings. Finally, creating more "natural" communicative utterances from the computer, rather than helping, exacerbates the problem by suggesting to the user that it is possible to engage in the dialogue with the system when it is not in fact the case.

In our early design study (study 3, Section IV), we found that if the programming language is constrained, novice programmers make far fewer errors. As noted earlier, this is perhaps unsurprising, but may suggest that visual languages derive some of their cognitive tractability from the limited power of expression of graphical representations, e.g. their difficulty in representing alternative possibilities [14].

Finally, when natural language is used for computational keywords, it would appear that novices require support both for understanding a particular computational concept, but also to determine the computational category to which it belongs (e.g. trigger, state, condition.)

VI. DESIGN GUIDELINES

We synthesized the findings from our three related studies to produce a set of design guidelines for developing novice programming environments that make use of natural language. Our focus is on how such environments can support young people to 1) create correct and complete computational rule specifications and 2) develop an understanding of computational concepts, and the skills to use them. The design guidelines are described below, along with a reference to the study (or studies) from which they arose (indicated as "S1" for study 1, etc.).

1. **Constrain the programming language:** novice programming languages, whether graphical or textual, should be constrained so as to minimize syntax errors. They should allow novices to choose from existing graphical blocks or textual keywords rather than requiring them to generate constructs from scratch. (S1, S2, S3);
2. **Delineate natural language from code:** where natural language is used, it should be clear to the user whether the language is "true" natural language (e.g. a conversation line in a story) or is being used as code. The use of graphics for code may be particularly useful as it provides an effective delineation of code from natural language (S2, S3);
3. **Highlight natural language used as a computational construct:** where natural language is used, highlight its computational function, e.g., computational keywords shown in a different font/colour in the case of a text-based language, or contained in blocks which clearly highlight the computational nature of the words used (S2, S3);
4. **Highlight distinctions between computational categories:** where natural language is used, in addition to point 3, provide support for computational category distinctions (e.g. *actions vs. state*), to avoid confusion between similar natural language phrases which have distinct computational properties (S3);
5. **Provide support for errors of omission/commission at the program composition phase**: support should be provided for both errors of omission and commission, thus ensuring that rules are completely and correctly specified. For example, program elements can be colour coded, or represented as different shapes. Similarly, certain combinations of blocks could be disallowed. If errors do persist at compilation, then any error messages should be brief and understandable. (S1, S2, S3);
6. **Use 'natural' natural language:** where natural language is used, it should be consistent with typical everyday use, as far as possible, avoiding unnatural syntax or phrasing (S2);
7. **Use natural language to convey information rather than engage in a dialogue**: care should be taken, when using natural language, to avoid giving the impression that the system is able to engage in a dialogue with the user, or more broadly, is more "intelligent" than it actually is. An environment that encourages a conversational style may actually exacerbate the formation of superbugs (S2);
8. **Provide support for debugging and collaboration:** use representations which allow the user, and other users, to easily understand the code they have written. Where programs are syntactically correct, but will not produce the behavior intended by the user, support should be provided to users so that they can review their code and find potential

semantic errors. In addition to enhancing individual comprehension and debugging, the representation should support collaborative programming efforts, allowing users to quickly become familiar with someone else's code. (S2).

The guidelines can be used in the design of new languages, and to analyse the properties of existing languages. The guidelines were implemented in Flip, a bi-modal (graphical and textual) language for game creation, described below.

VII. The Flip Languge

A. An overview of Flip

Flip is designed for use with the NWN2 toolset, and allows novice programmers to compose scripts by dragging and dropping graphical blocks into a pre-existing framework. The interface also includes a full natural language description of the script under construction: as the user adds blocks to their program, the natural language description updates dynamically. Fig. 4 shows the layout of the Flip interface.

The **Block Box** contains the blocks used to create scripts, organised by computational category and object type. Selecting a category from the top panel highlights it, and displays all blocks in that category below. Blocks represent computational concepts (actions, conditions, states), and are colour-coded by type. Blocks have slots that must be filled by objects of a certain type. The slots are similarly colour-coded to indicate which types of objects they can receive, and empty slots indicate, in natural language, the type of slot filler required.

The **Spine** is where the script is composed, by attaching blocks to the silver pegs. The spine can be extended indefinitely to accommodate additional actions, and scripts execute in a top down manner. In Fig. 4, a control block has been added to the spine, which in turn has its own spine for actions to be carried out if the specified condition is true.

The **Event Slot** takes a single event block, which dictates when the script will execute.

The **Natural Language (or 'plain English') Box** shows a natural language description of the script under creation. It is automatically generated, and dynamically updated every time the user makes a change to their script. The natural language box gives a full description of the script meaning, explaining in detail what will happen and under what conditions. Whilst the words on the blocks are designed for brevity and to reflect the most common way of describing the underlying computational concepts, the natural language description is more complete, and uses terms that our design work indicated would be understood by our target users.

B. Design guidelines embodied in Flip

1. **Constrain the programming language:** achieved through the use of a set of graphical code blocks, eliminating the need to type code from scratch. Also achieved by making the natural language component of the language non-editable, thus preventing the user from inadvertently introducing syntax errors;

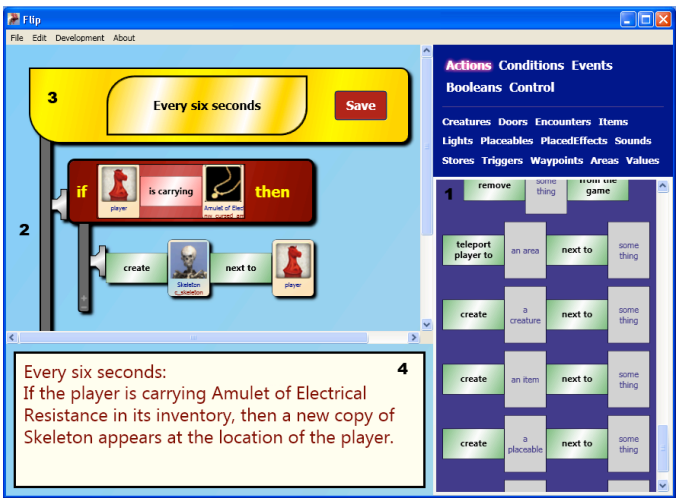

Fig. 4 Flip interface – 1: Block Box, 2: Spine, 3: Event Slot, 4: Natural Language Box

2. **Delineate natural language from code:** Code appears in Flip's graphical code composition interface, while its natural language equivalent appears in the plain English box. Natural language used in the script (e.g. messages to be displayed to the player) is shown as a separate, colour-coded block within the graphical language, and in quotes in the plain English box;
3. **Highlight natural language used as a computational construct:** Flip uses correct computational terms to represent the underlying concepts on all interface menus and graphical programming blocks, allowing young people to begin to learn and use the "language of computation" [4]. These computational keywords are automatically translated into fuller, everyday language in the plain English box, further distinguishing computational language from natural language (reinforcing point 2);
4. **Highlight distinctions between computational categories:** blocks are visually organized and colour coded by computational category (*actions*, *conditions*) to avoid confusion between terms that are similar from a natural language perspective ("opens" vs. "is open") but computationally distinct (action vs. state);
5. **Provide support for errors of omission/commission at the program composition phase**: blocks and slot fillers are differentiated by colour and shape, with colour used to link blocks with their corresponding slot fillers. This feature aims to support users' understanding of differing types of command, and to help them avoid errors of commission. A further feature prevents blocks of the wrong type from snapping into place, thus preventing type errors. Errors of omission that prevent the script from compiling trigger a short error message (see point 7);
6. **Use 'natural' natural language:** Flip's "plain English" box is designed to be as similar as possible to spoken English (e.g. when describing one's script to another person), avoiding any unusual turns of phrase. Phrasings were derived, as much as possible, from the corpus of rule descriptions gathered from target users (see Section II.B);

7. **Use natural language to convey information rather than engage in a dialogue**: Written error messages from the system to the user are rare, as syntax errors are prevented. The few error messages that are necessary (when a crucial element has been omitted) have been kept very short, in order to avoid suggesting that the compiler a human-like capacity for language comprehension and dialogue;
8. **Provide support for debugging and collaboration:** Support for user identification of semantic errors is provided by the plain English box. For users who are struggling to express their ideas within the formal constraints of the programming language, it allows them to view their script in a familiar format, supporting both comprehension (checking the meaning of the script they are working on) and debugging (figuring out why the script is not correct and fixing it). As the plain English box allows learners to read a description in language similar to how they might describe it to a peer, this should also allow for more effective collaboration, as users will quickly be able to get up to speed with each other's programs, and help to problem solve.

VIII. Evaluating the design guidelines in Flip

Flip has been evaluated in a number of empirical studies in different settings, including an investigation on whether its use led to an improvement in young people's computational abilities in other contexts (Howland and Good, 2015). Here we describe two studies that considered the extent to which Flip is usable by our target users for program generation, and supports an understanding of computation. We were also interested in how the design guidelines embodied in Flip contributed to the above two points, i.e. whether they contribute to Flip's usability and provide support for program generation and computational understanding.

The evaluation studies were carried out in two different real world contexts, and with different aims. Study one was an observational study that examined, in detail, the use of Flip in a secondary school classroom over a 2 hour lesson. Study two was a longitudinal study looking at the use of Flip over the course of a complete game creation project.

A. Flip observation study

1) Method

Twenty-one pupils aged 11-12 (9 male, 12 female) took part in the study, which took place in a secondary school in Scotland. As part of their previous year's IT lessons, twenty of the pupils had taken part in a game creation project (lasting approximately 8 weeks). The project had used the Electron toolset, but not the Flip language, so the pupils had previous knowledge of game creation, but no experience of Flip.

Pupils were told that they would be testing and giving feedback on a new language, designed to work with the game creation tool they had previously used, and to make it easier to create more complex game events. Pupils worked in pairs (with 1 group of 3) to create a game over the course of one hour. Pupils were initially shown a short video demonstrating how to add a script that is triggered by the speaking of a conversation line, and were asked to try adding a similar script to their games. After 20 minutes, they were shown a second video demonstrating how to trigger scripts using other events. After a further 20 minutes of working on their games, pupils were given a final demonstration on how to use control blocks to add conditionals to their scripts.

Four researchers were present in the classroom along with the teacher. One researcher gave the demonstrations and took prime responsibility for answering pupils' questions, while the other three researchers acted primarily as observers. The researchers made a number of video recordings of onscreen interactions. Log files and scripts were also collected for analysis. Researcher notes were written up after the session, and the video footage was transcribed and analysed.

2) Results

a) Flip usage stats and overall usability

All participants succeeded in creating at least one full script with an event and action(s), while seven of the ten pairs successfully added conditionals to their scripts. Pupils commented that Flip represented a definite improvement in terms of allowing them to write their own scripts, with one pair noting excitedly that Flip was "really good!"

b) Flip usage and understanding

Action blocks were well understood, with no instances of confusion over attaching action blocks to the spine. Control blocks seemed to cause problems for some pairs. Participants selected appropriate control blocks, but they sometimes chose the wrong type of block to complete the conditional slot. Most pairs seemed to understand the "Then" spine intuitively, and were able to complete it without additional help or discussion, but one tried to click on the word "Then" to add their actions.

Events seemed to cause problems for a few pupils. Although event blocks were a different shape and colour to other types of block, the four instances of incomplete scripts that we observed were due to the lack of an event block. In one case, a user tried to save a script without an event, and received an error message, then quickly added an event and successfully saved it. In another case, the pair did not initially know what to put in the event block, but were able to add an event and save their script with help from their peers. Two similar cases were observed where one user thought the script was ready to save, but their partner pointed out the missing event.

There were some instances of initial confusion around the distinction between different computational categories (actions, conditions, etc.), similar to the design study, but this appeared to be limited to instances involving the conditional slot of control blocks. A few users tried to drag the wrong type of block into the conditional slot, but could tell that something was wrong, as the block would not snap into place. This caused momentary frustration, but led to them trying another block type, with eventual success. In one case, after attempting to add an action to a slot requiring a condition, one participant said to her partner "we need something that's that colour", pointing to the pink inside the slot. On switching to the 'Conditions' menu, they immediately noted that condition blocks were similarly pink, and successfully completed their control block. The same issue was observed with another pair, with a similar method of resolution, but without an explicit discussion of colour.

c) Use of the natural language box

The natural language description was used as a sense checking mechanism on numerous occasions, with ten recorded cases of participants reading the natural language aloud, before deciding whether it described their intentions accurately. In two of these cases, this led to a revision of the script, while in the remaining cases, it gave participants the confidence to move on to testing the event in game. There were two examples of pupils reading the natural language description and paraphrasing it in order to explain the code to either a researcher or a peer. There are likely to have been a number of other cases where pupils read the natural language description silently to themselves, but as this study did not employ eye-tracking methods this cannot be accurately reported on.

Overall, there were many examples of pupils discussing computational concepts, including conditionals, with each other by the end of the session, an observation that is supported by a longitudinal study in another school in which the teacher noted that Flip provides pupils with a language for expressing computational concepts [4].

B. Flip longitudinal study

1) Method

14 young people aged 11-15 (1 female, 13 male) took part in a 4-day game creation workshop. Participation was voluntary, in response to advertisements for holiday activities. Three participants had used the game creation toolset, but without Flip (two from our previous workshops, and one from an unconnected workshop). Ten participants reported no prior programming experience, one was not sure, while the remaining three had very limited experience.

Participants attended the workshop over four days from 10am to 4pm. With a 45 minute lunch break per day, and an hour spent on related activities at the beginning and end of the workshop respectively, participants spent approximately 19 hours on the game creation project.

Demonstrations by the authors introduced participants to the key functionality of the toolset and Flip. At the end of the workshop, semi-structured interviews were conducted with all participants, which included general questions about game creation, and six questions focusing specifically on Flip. Screenshots of the Flip interface were used as prompts during the interview, and interviews were audio recorded and later transcribed. Copies of all modules created and scripts written were also collected, as well as log files for each participant.

2) Results

a) Flip usage stats and overall usability

All workshop participants successfully created fully functional scripts. There were 410 script saves recorded, and a total of 260 individual scripts (a mean of 18.57 scripts per participant). A total of 780 actions were used in the scripts (a mean of 3 actions per script), and 260 events (as every script is triggered by an event). 71% of participants included conditionals in their scripts, with 73 conditionals used in total.

All of the participants noted that they found Flip easy to use, with one participant stating that it was because "it was very straight forward and the words aren't too complicated to understand", and another noting that "…it's easier than the proper one that came with the toolset".

b) Flip usage and understanding

During the interviews, we asked participants to explain various components of Flip in order to gauge their understanding. When asked about the event block, 12 participants gave a clear explanation of its purpose, while two were able to provide example of events, but could not move beyond the specifics to a more abstract explanation. When asked about control blocks, specifically an If…Then… block, 10 participants gave a clear explanation of the way the block functions, with some able to describe it using computational terms, and others expressing the meaning in more non-standard ways: "… It's an If… err, equation, so like…if whatever variable you specify is…is one way, then it will do… do whatever it is in the script, but if it isn't, then it won't." The remaining four participants seemed to understand the purpose generally, but could not find the language to express it unambiguously.

c) Flip and natural language

When asked to explain the purpose of the natural language box, eleven people gave clear, high-level explanations, while the remaining three appeared to understand its purpose, but focussed on specific examples. Seven people noted that they considered the natural language description to be a simplified explanation for when they did not understand the blocks above, with some mentioning that it was particularly useful for complex scripts. Three people described it as a way of checking that the script would work as intended, and spotting where corrections were necessary.

IX. Design Guidelines and Novice Support

When looking at the findings from the two evaluations in light of the design guidelines, the main benefits in terms of support centred around issues of constraint, a clear separation between code and natural language, support for errors, and use of natural language for comprehension and debugging.

In brief, syntax errors were not observed due to the highly constrained nature of the language (DG1), with support built in to the graphical language and the environment as a whole (DG4, DG5). The non-editable natural language representation also prevented the introduction of syntax errors (DG2). The natural language representation seemed to be understandable to pupils (DG6) and to successfully support individual and collaborative code comprehension and debugging (DG8). It also acted as a support for communication about computation, with pupils using it to explain their code to others.

Overall, the empirically derived design guidelines allowed us, in the first instance, to determine how to best incorporate natural language into the environment in a way that would be helpful to novice programmers, rather than hindering their progress or even leading to further misconceptions. In turn, reviewing the empirical evaluations of Flip in light of the design guidelines allowed us to see which features were having an impact on program generation, comprehension and debugging, and in what ways.

REFERENCES

[1] Good, J., K. Howland, and K. Nicholson. Young People's Descriptions of Computational Rules in Role-Playing Games: An Empirical Study. in 2010 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC). 2010. IEEE.

[2] Good, J. and J. Oberlander, Verbal effects of visual programs: Information type, structure and error in program summaries. Document Design, 2002. 3(2): p. 120-134.

[3] Good, J. and J. Robertson. Learning and Motivational Affordances in Narrative-based Game Authoring. in Narrative and Interactive Learning Environments, NILE 2006. 2006. Edinburgh, Scotland.

[4] Howland, K. and J. Good, Learning to communicate computationally with Flip: A bi-modal programming language for game creation. Computers & Education, 2015. 80(0): p. 224-240.

[5] Howland, K., J. Good, and J. Robertson, A learner-centred design approach to developing a visual language for interactive storytelling. Proceedings of the 6th international conference on Interaction design and children, 2007: p. 45-52.

[6] Kelleher, C. and R. Pausch, Lowering the barriers to programming. ACM Computing Surveys, 2005. 37(2): p. 83-137.

[7] Miller, L. Naive programmer problems with specification of transfer-of-control. in Proceedings of National Computer Conference, AMPS 44. 1975. ACM.

[8] Miller, L. Behavioral Studies of the Programming Process. Research Report. 1978

[9] Nelson, G. Natural Language, Semantics Analysis and Interactive Fiction. 2006; Available from: http://www.informfiction.org/I7Downloads/Documents/WhitePaper.pdf. Accessed March 12, 2010.

[10] Pea, R.D., Language-independent conceptual" bugs" in novice programming. Journal of Educational Computing Research, 1986. 2(1): p. 25-36.

[11] Robertson, J. and J. Good, Children's narrative development through computer game authoring. TechTrends, 2005. 49(5): p. 43-59.

[12] Robertson, J. and J. Good, Story Creation in Virtual Game Worlds, in Communications of the ACM2005. p. 61-65.

[13] Sammet, J.E., The use of English as a programming language, in Communications of the ACM1966. p. 228-230.

[14] Stenning, K. and J. Oberlander, A cognitive theory of graphical and linguistic reasoning: logic and implementation. Cognitive science, 1995. 19(1): p. 97-140.

[15] Whitley, K.N., Visual programming languages and the empirical evidence for and against. Journal of Visual Languages & Computing, 1997. 8(1): p. 109-142.